

NISP Toolbox Manual

Michael Baudin (EDF)
Jean-Marc Martinez (CEA)

Version 0.4
February 2013

Abstract

This document is an introduction to the NISP module v2.5. We present the installation process of the module in binary from ATOMS or from the sources. Then we present the configuration functions and the `randvar`, `setrandvar` and `polychaos` classes. Several examples are provided for each class, which provides an overview of the use of NISP in practical situations.

Contents

1	Introduction	4
1.1	The OPUS project	4
1.2	The NISP library	4
1.3	The NISP module	5
1.4	Installing the toolbox from ATOMS	7
1.5	Configuration functions	10
2	The randvar class	11
2.1	The distribution functions	11
2.1.1	Overview	11
2.1.2	The Log-Normal distribution	12
2.1.3	The Log-Uniform distribution	12
2.1.4	Uniform random number generation	13
2.2	Methods	13
2.2.1	Overview	13
2.2.2	The Oriented-Object system	13
2.3	Examples	16
2.3.1	A sample session	16
2.3.2	Variable transformations	16
3	The setrandvar class	21
3.1	Introduction	21
3.2	Examples	21
3.2.1	A Monte-Carlo design with 2 variables	21
3.2.2	A Monte-Carlo design with 2 variables	25
3.2.3	A LHS design	27
3.2.4	A note on the LHS samplings	31
3.2.5	Other types of DOEs	34
4	The polychaos class	38
4.1	Introduction	38
4.2	Examples	38
4.2.1	Product of two random variables	38
4.2.2	A note on performance	43
4.2.3	The Ishigami test case	45
5	Thanks	49

Bibliography	50
A Installation	51
A.1 Architecture of the toolbox	51
A.2 Installing the toolbox from the sources	51

Copyright © 2012-2013 - Michael Baudin
Copyright © 2008-2011 - INRIA - Michael Baudin
Copyright © 2008-2011 - CEA - Jean-Marc Martinez

This file must be used under the terms of the Creative Commons Attribution-ShareAlike 3.0
Unported License:

<http://creativecommons.org/licenses/by-sa/3.0>

Chapter 1

Introduction

1.1 The OPUS project

The goal of this toolbox is to provide a tool to manage uncertainties in simulated models. This toolbox is based on the NISP library, where NISP stands for "Non-Intrusive Spectral Projection". This work has been realized in the context of the OPUS project,

<http://opus-project.fr>

"Open-Source Platform for Uncertainty treatments in Simulation", funded by ANR, the french "Agence Nationale pour la Recherche":

<http://www.agence-nationale-recherche.fr>

The toolbox is released under the Lesser General Public Licence (LGPL), as all components of the OPUS project.

This module was presented in the "42èmes Journées de Statistique, du 24 au 28 mai 2010" [2].

1.2 The NISP library

The NISP library is based on a set of 3 C++ classes so that it provides an object-oriented framework for uncertainty analysis. The Scilab toolbox provides a pseudo-object oriented interface to this library, so that the two approaches are consistent. The NISP library is release under the LGPL licence.

The NISP library provides three tools, which are detailed below.

- The "randvar" class allows to manage random variables, specified by their distribution law and their parameters. Once a random variable is created, one can generate random numbers from the associated law.
- The "setrandvar" class allows to manage a collection of random variables. This collection is associated with a sampling method, such as MonteCarlo, Sobol, Quadrature, etc... It is possible to build the sample and to get it back so that the experiments can be performed.
- The "polychaos" class allows to manage a polynomial representation of the simulated model. One such object must be associated with a set of experiments which have been performed.

This set may be read from a data file. The object is linked with a collection of random variables. Then the coefficients of the polynomial can be computed by integration (quadrature). Once done, the mean, the variance and the Sobol indices can be directly computed from the coefficients.

The figure 1.1 presents the NISP methodology. The process requires that the user has a numerical solver, which has the form $Y = f(X)$, where X are input uncertain parameters and Y are output random variables. The method is based on the following steps.

- We begin by defining normalized random variables ξ . For example, we may use a random variables in the interval $[0, 1]$ or a Normal random variable with mean 0 and variance 1. This choice allows to define the basis for the polynomial chaos, denoted by $\{\Psi_k\}_{k \geq 0}$. Depending on the type of random variable, the polynomials $\{\Psi_k\}_{k \geq 0}$ are based on Hermite, Legendre or Laguerre polynomials.
- We can now define a Design Of Experiments (DOE) and, with random variable transformations rules, we get the physical uncertain parameters X . Several types of DOE are available: Monte-Carlo, Latin Hypercube Sampling, etc... If N experiments are required, the DOE define the collection of normalized random variables $\{\xi_i\}_{i=1,N}$. Transformation rules allows to compute the uncertain parameters $\{X_i\}_{i=1,N}$, which are the input of the numerical solver f .
- We can now perform the simulations, that is compute the collection of outputs $\{Y_i\}_{i=1,N}$ where $Y_i = f(X_i)$.
- The variables Y are then projected on the polynomial basis and the coefficients y_k are computed by integration or regression.

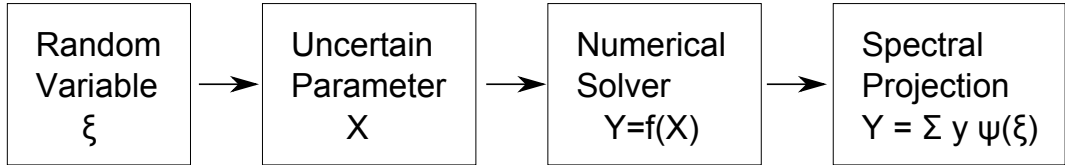


Figure 1.1: The NISP methodology

1.3 The NISP module

The NISP toolbox is available under the following operating systems:

- Linux 32 bits,
- Linux 64 bits,
- Windows 32 bits,
- Mac OS X.

The following list presents the features provided by the NISP toolbox.

- Manage various types of random variables:
 - uniform,
 - normal,
 - exponential,
 - log-normal.
- Generate random numbers from a given random variable,
- Transform an outcome from a given random variable into another,
- Manage various Design of Experiments for sets of random variables,
 - Monte-Carlo,
 - Sobol,
 - Latin Hypercube Sampling,
 - various samplings based on Smolyak designs.
- Manage polynomial chaos expansion and get specific outputs, including
 - mean,
 - variance,
 - quantile,
 - correlation,
 - etc...
- Generate the C source code which computes the output of the polynomial chaos expansion.

This User's Manual completes the online help provided with the toolbox, but does not replace it. The goal of this document is to provide both a global overview of the toolbox and to give some details about its implementation. The detailed calling sequence of each function is provided by the online help and will not be reproduced in this document. The inline help is presented in the figure 1.2.

For example, in order to access to the help associated with the **randvar** class, we type the following statements in the Scilab console.

```
help randvar
```

The previous statements opens the Help Browser and displays the helps page presented in figure

Several demonstration scripts are provided with the toolbox and are presented in the figure 1.4. These demonstrations are available under the "?" question mark in the menu of the Scilab console.

Finally, the unit tests provided with the toolbox cover all the features of the toolbox. When we want to know how to use a particular feature and do not find the information, we can search in the unit tests which often provide the answer. See in the section A.1 for details on the internal structure of the toolbox.

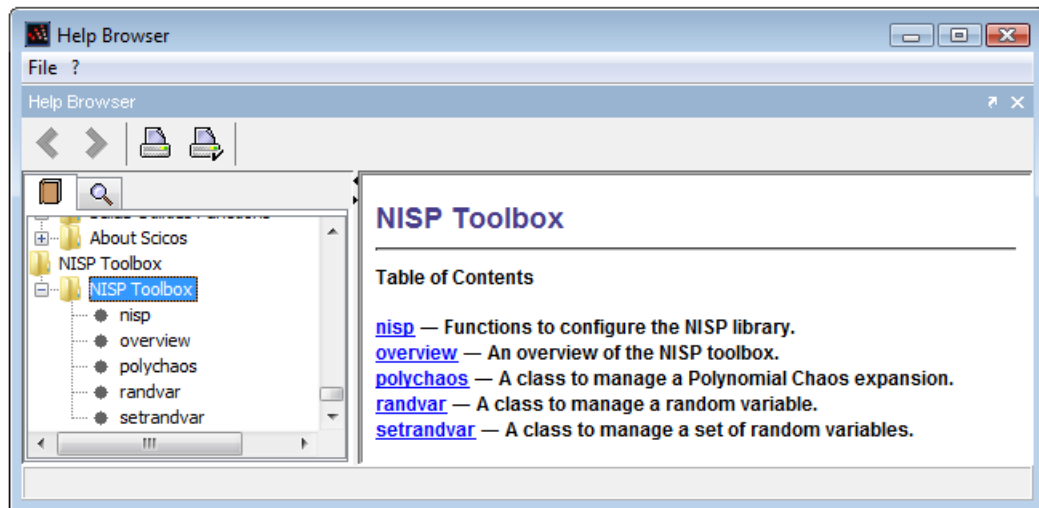


Figure 1.2: The NISP inline help.

1.4 Installing the toolbox from ATOMS

There are two possible ways of installing the NISP toolbox in Scilab:

- use the ATOMS system and get a binary version of the toolbox,
- build the toolbox from the sources.

In this section, we present the method to install NISP from ATOMS. The installation of the toolbox from the sources is presented in appendix, in section A.2.

The ATOMS component is the Scilab tool which allows to search, download, install and load toolboxes. ATOMS comes with Scilab v5.2. The Scilab-NISP toolbox has been packaged and is provided mainly by the ATOMS component. The toolbox is provided in binary form, depending on the user's operating system. The Scilab-NISP toolbox is available for the following platforms:

- Windows 32 bits, 64 bits,
- Linux 32 bits, 64 bits,
- Mac OS X.

The ATOMS component allows to use a toolbox based on compiled source code, without having a compiler installed in the system.

Installing the Scilab-NISP toolbox from ATOMS requires to run the `atomsInstall` function, then to restart Scilab.

In the following Scilab session, we use the `atomsInstall()` function to download and install the binary version of the toolbox corresponding to the current operating system.

```
-->atomsInstall ( "NISP" )
ans =
!NISP 2.1 allusers D:\Programs\SC3623~1\contrib\NISP\2.1 I !
```

The "allusers" option of the `atomsInstall` function can be used to install the toolbox for all the users of this computer. Then we restart Scilab, and the NISP toolbox is automatically loaded.

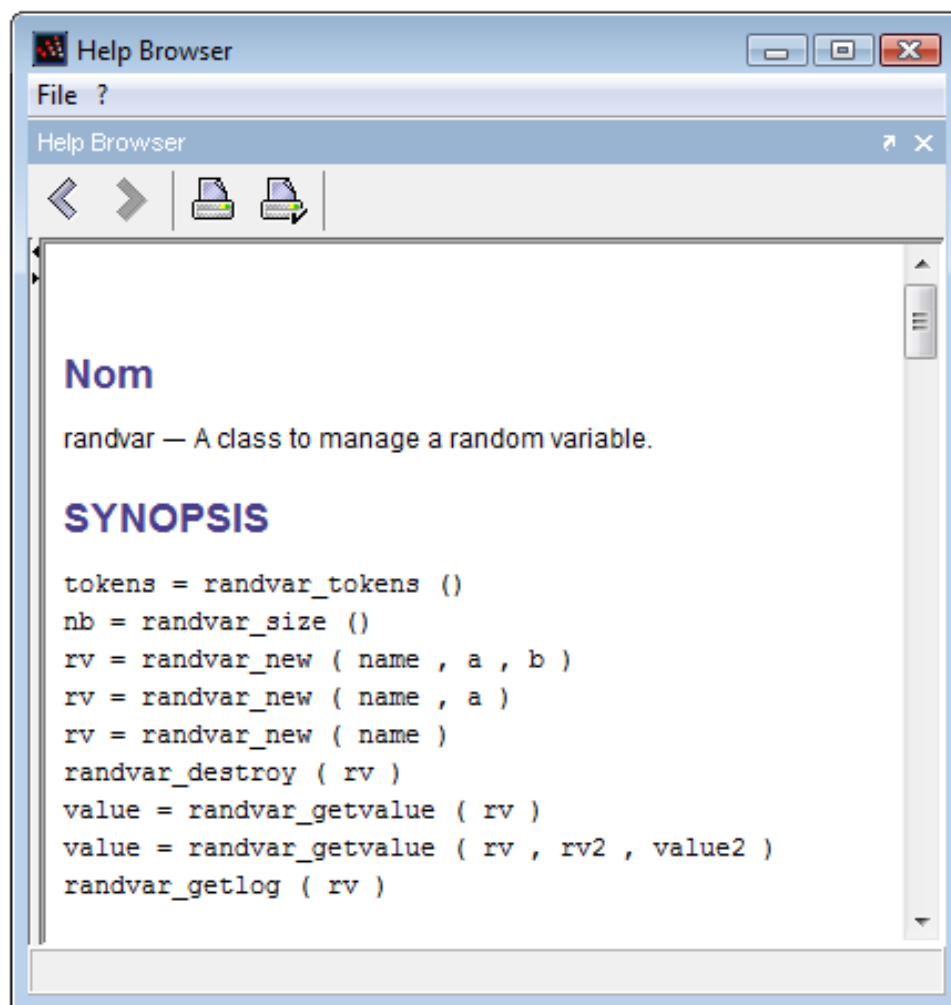


Figure 1.3: The online help of the randvar function.

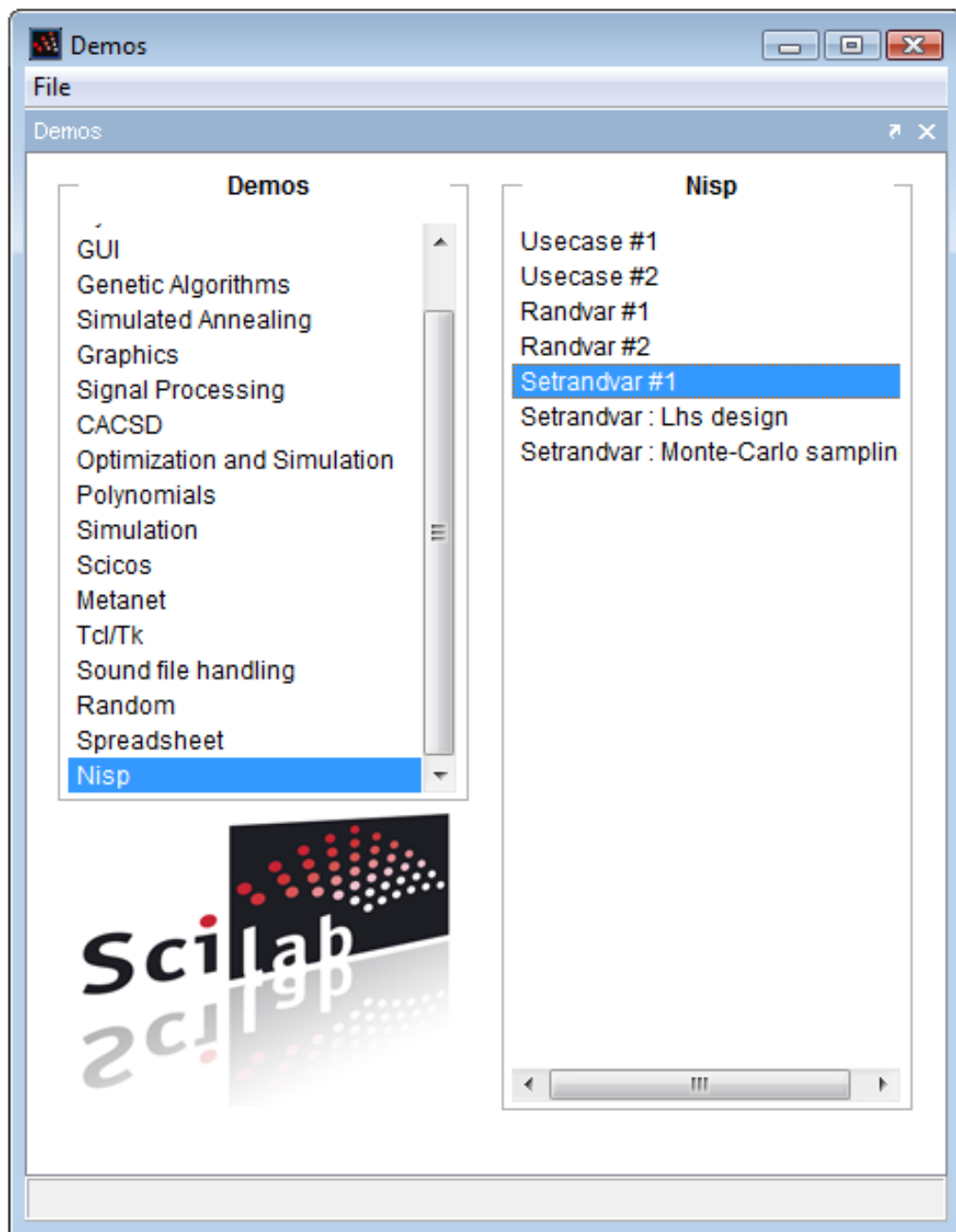


Figure 1.4: Demonstrations provided with the NISP toolbox.

1.5 Configuration functions

In this section, we present functions which allow to configure the NISP toolbox.

The `nisp_*` functions allows to configure the global behaviour of the toolbox. These functions allows to startup and shutdown the toolbox and initialize the seed of the random number generator. They are presented in the figure 1.5.

Public functions	
<code>level=nisp_verboselevelget()</code>	Returns the current verbose level.
<code>nisp_verboselevelset(level)</code>	Sets the value of the verbose level.
<code>nisp_initseed(seed)</code>	Sets the seed of the uniform NISP random number generator.
<code>nisp_destroyall</code>	Destroy all current NISP objects.
<code>nisp_getpath</code>	Returns the path to the current module.
<code>nisp_printall</code>	Prints all current NISP objects.
Private functions	
<code>nisp_startup()</code>	Starts up the NISP toolbox.
<code>nisp_shutdown()</code>	Shuts down the NISP toolbox.

Figure 1.5: Outline of the configuration methods.

The `nisp_initseed ( seed )` is useful when we want to have reproducible results. It allows to set the seed of the generator at a particular value, so that the sequence of uniform pseudo-random numbers is deterministic. When the toolbox is started up, the seed is automatically set to 0, which allows to get the same results from session to session.

The user has no need to explicitly call the `nisp_startup()` and `nisp_shutdown()` functions. Indeed, these functions are called automatically by the *etc/NISP.start* and *etc/NISP.quit* scripts, located in the toolbox directory structure. See the section A.1 for details on this topic.

Chapter 2

The randvar class

In this section, we present the **randvar** class, which allows to define a random variable, and to generate random numbers from a given distribution function.

2.1 The distribution functions

In this section, we present the distribution functions provided by the **randvar** class. We especially present the Log-normal distribution function.

2.1.1 Overview

The table 2.1 gives the list of distribution functions which are available with the **randvar** class [3].

Each distribution functions have zero, one or two parameters. One random variable can be specified by giving explicitly its parameters or by using default parameters. The parameters for all distribution function are presented in the figure 2.2, which also presents the conditions which must be satisfied by the parameters.

Name	$f(x)$	$E(X)$	$V(X)$
"Normale"	$\frac{1}{2\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right)$	μ	σ^2
"Uniforme"	$\begin{cases} \frac{1}{b-a}, & \text{if } x \in [a, b] \\ 0 & \text{if } x \notin [a, b] \end{cases}$	$\frac{b+a}{2}$	$\frac{(b-a)^2}{12}$
"Exponentielle"	$\begin{cases} \lambda \exp(-\lambda x), & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$
"LogNormale"	$\begin{cases} \frac{1}{\sigma x \sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(\ln(x)-\mu)^2}{\sigma^2}\right), & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$	$\exp\left(\mu + \frac{\sigma^2}{2}\right)$	$(\exp(\sigma^2) - 1) \exp(2\mu + \sigma^2)$
"LogUniforme"	$\begin{cases} \frac{1}{x(b-a)}, & \text{if } x \in [\exp(a), \exp(b)] \\ 0 & \text{otherwise.} \end{cases}$	$\frac{\exp(b)-\exp(a)}{b-a}$	$\frac{1}{2} \frac{\exp(b)^2 - \exp(a)^2}{b-a} - E(x)^2$

Figure 2.1: Distributions functions of the **randvar** class. – The expected value is denoted by $E(X)$ and the variance is denoted by $V(X)$.

Name	Parameter #1 : a	Parameter #2 : b	Conditions
"Normale"	$\mu = 0$	$\sigma = 1$	$\sigma > 0$
"Uniforme"	$a = 0$	$b = 1$	$a < b$
"Exponentielle"	$a = 1$	-	-
"LogNormale"	$\mu = 0$	$\sigma = 1$	$\sigma > 0$
"LogUniforme"	$a = 0$	$b = 1$	$a < b$

Figure 2.2: Default parameters for distributions functions.

2.1.2 The Log-Normal distribution

A log-normal distribution is a probability distribution of a random variable whose logarithm is normally distributed. If X is a random variable with a normal distribution, then $Y = \exp(X)$ has a log-normal distribution.

The "LogNormale" law is defined by the expected value μ and the standard deviation σ of the underlying Normal random variable. In other words, the random variable $\log(X)$ has mean μ and variance σ^2 . The expected value and the variance of the Log Normal law are given by

$$E(X) = \exp\left(\mu + \frac{1}{2}\sigma^2\right) \quad (2.1)$$

$$V(X) = (\exp(\sigma^2) - 1) \exp(2\mu + \sigma^2). \quad (2.2)$$

It is possible to invert these formulas, in the situation where the given parameters are the expected value and the variance of the Log Normal random variable. We can invert completely the previous equations and get

$$\mu = \ln(E(X)) - \frac{1}{2} \ln\left(1 + \frac{V(X)}{E(X)^2}\right) \quad (2.3)$$

$$\sigma^2 = \ln\left(1 + \frac{V(X)}{E(X)^2}\right). \quad (2.4)$$

In particular, the expected value μ of with the Normal random variable satisfies the equation

$$\mu = \ln(E(X)) - \sigma^2. \quad (2.5)$$

Caution ! These are the parameters for NISP v2.5. Earlier versions of the toolbox used other parameters. See the changelog.txt file in your toolbox to see how to update your scripts.

2.1.3 The Log-Uniform distribution

A log-uniform distribution is a probability distribution of a random variable whose logarithm has a uniform distribution. If X is a random variable with a uniform distribution, then $Y = \exp(X)$ has a log-uniform distribution.

The "LogUniforme" law is defined by the minimum a and the maximum b of the underlying Uniform random variable. In other words, the random variable $\log(X)$ has minimum a and maximum b .

If X is a LogUniforme random variable with minimum A and maximum B , then:

$$a = \log(A) \quad (2.6)$$

$$b = \log(B) \quad (2.7)$$

Caution ! These are the parameters for NISP v2.5. Earlier versions of the toolbox used other parameters. See the changelog.txt file in your toolbox to see how to update your scripts.

2.1.4 Uniform random number generation

In this section, we present the generation of uniform random numbers.

Since v2.5, the library uses the Mersenne-Twister pseudo-random number generator. This generator is from M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January, pp.3-30 1998.

2.2 Methods

In this section, we give an overview of the methods which are available in the **randvar** class.

2.2.1 Overview

The figure 2.3 presents the methods available in the **randvar** class. The inline help contains the detailed calling sequence for each function and will not be repeated here.

Constructors
<code>rv = randvar_new (type [, options])</code>
Methods
<code>value = randvar_getvalue (rv [, options])</code>
<code>randvar_getlog (rv)</code>
Destructor
<code>randvar_destroy (rv)</code>
Static methods
<code>rvlist = randvar_tokens ()</code>
<code>nbrv = randvar_size ()</code>

Figure 2.3: Outline of the methods of the **randvar** class.

2.2.2 The Oriented-Object system

In this section, we present the token system which allows to emulate an oriented-object programming with Scilab. We also present the naming convention we used to create the names of the functions.

The **randvar** class provides the following functions.

- The constructor function `randvar_new` allows to create a new random variable and returns a *token* `rv`.
- The method `randvar_getvalue` takes the token `rv` as its first argument. In fact, all methods takes as their first argument the object on which they apply.
- The destructor `randvar_destroy` allows to delete the current object from the memory of the library.
- The static methods `randvar_tokens` and `randvar_size` allows to query the current object which are in use. More specifically, the `randvar_size` function returns the number of current `randvar` objects and the `randvar_tokens` returns the list of current `randvar` objects.

In the following Scilab sessions, we present these ideas with practical uses of the toolbox.

Assume that we start Scilab and that the toolbox is automatically loaded. At startup, there are no objects, so that the `randvar_size` function returns 0 and the `randvar_tokens` function returns an empty matrix.

```
-->nb = randvar_size()
nb =
    0.
-->tokenmatrix = randvar_tokens()
tokenmatrix =
    []
```

We now create 3 new random variables, based on the Uniform distribution function. We store the tokens in the variables `vu1`, `vu2` and `vu3`. These variables are regular Scilab double precision floating point numbers. Each value is a token which represents a random variable stored in the toolbox memory space.

```
-->vu1 = randvar_new("Uniforme")
vu1 =
    0.
-->vu2 = randvar_new("Uniforme")
vu2 =
    1.
-->vu3 = randvar_new("Uniforme")
vu3 =
    2.
```

There are now 3 objects in current use, as indicated by the following statements. The `tokenmatrix` is a row matrix containing regular double precision floating point numbers.

```
-->nb = randvar_size()
nb =
    3.
-->tokenmatrix = randvar_tokens()
tokenmatrix =
    0.    1.    2.
```

We assume that we have now made our job with the random variables, so that it is time to destroy the random variables. We call the `randvar_destroy` functions, which destroys the variables.

```
-->randvar_destroy(vu1);
-->randvar_destroy(vu2);
-->randvar_destroy(vu3);
```

We can finally check that there are no random variables left in the memory space.

```
-->nb = randvar_size()
nb =
    0.
-->tokenmatrix = randvar_tokens()
tokenmatrix =
    []
```

Scilab is a wonderful tool to experiment algorithms and make simulations. It happens sometimes that we are managing many variables at the same time and it may happen that, at some point, we are lost. The static methods provides tools to be able to recover from such a situation without closing our Scilab session.

In the following session, we create two random variables.

```
-->vu1 = randvar_new("Uniforme")
vu1 =
    3.
-->vu2 = randvar_new("Uniforme")
vu2 =
    4.
```

Assume now that we have lost the token associated with the variable `vu2`. We can easily simulate this situation, by using the `clear` function, which destroys a variable from Scilab's memory space.

```
-->clear vu2
-->randvar_getvalue(vu2)
                                !--error 4
Undefined variable: vu2
```

It is now impossible to generate values from the variable `vu2`. Moreover, it may be difficult to know exactly what went wrong and what exact variable is lost. At any time, we can use the `randvar_tokens` function in order to get the list of current variables. Deleting these variables allows to clean the memory space properly, without memory loss.

```
-->randvar_tokens()
ans =
    3.    4.
-->randvar_destroy(3)
ans =
    3.
-->randvar_destroy(4)
ans =
    4.
-->randvar_tokens()
ans =
    []
```

2.3 Examples

In this section, we present to examples of use of the `randvar` class. The first example presents the simulation of a Normal random variable and the generation of 1000 random variables. The second example presents the transformation of a Uniform outcome into a LogUniform outcome.

2.3.1 A sample session

We present a sample Scilab session, where the `randvar` class is used to generate samples from the Normale law.

In the following Scilab session, we create a Normale random variable and compute samples from this law. The `nisp_initseed` function is used to initialize the seed for the uniform random variable generator. Then we use the `randvar_new` function to create a new random variable from the Normale law with mean 1. and standard deviation 0.5. The main loop allows to compute 1000 samples from this law, based on calls to the `randvar_getvalue` function. Once the samples are computed, we use the Scilab function `mean` to check that the mean is close to 1 (which is the expected value of the Normale law, when the number of samples is infinite). Finally, we use the `randvar_destroy` function to destroy our random variable. Once done, we plot the empirical distribution function of this sample, with 50 classes.

```
nisp_initseed ( 0 );
mu = 1.0;
sigma = 0.5;
rv = randvar_new("Normale" , mu , sigma);
nbshots = 1000;
values = zeros(nbshots);
for i=1:nbshots
    values(i) = randvar_getvalue(rv);
end
mymean = mean (values);
mysigma = st_deviation(values);
myvariance = variance (values);
mprintf("Mean_:_%f_(expected=_%f)\n", mymean, mu);
mprintf("Std._dev._:_%f_(expected=_%f)\n", mysigma, sigma);
mprintf("Variance_:_%f_(expected=_%f)\n", myvariance, sigma^2);
randvar_destroy(rv);
histplot(50,values)
xtitle("Histogram_of_X","X","P(x)")
```

The previous script produces the following output.

```
Mean : 0.988194 (expected = 1.000000)
Std. dev. : 0.505186 (expected = 0.500000)
Variance : 0.255213 (expected = 0.250000)
```

The previous script also produces the figure [2.4](#).

2.3.2 Variable transformations

In this section, we present the transformation of a random variable with given distribution into a variable with another distribution. Then we present some of the many the transformations which

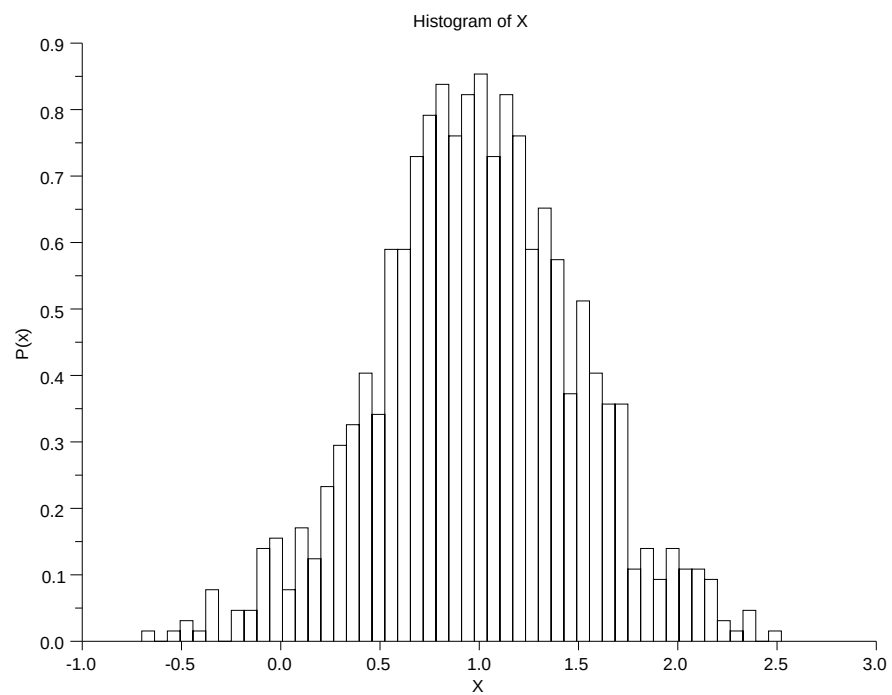


Figure 2.4: The histogram of a Normal random variable with 1000 samples.

are provided by the library.

Any outcome from a random variable x can be transformed into the outcome of another variable y . This can be done by computing $u = F_X(x)$, where F_X is the cumulated distribution function of X . This transforms the random variable X into the random variable U , which has a uniform distribution in $[0, 1]$. Then we invert the CDF, by computing $y = F_Y^{-1}(u)$, where F_Y is the cumulated distribution function of Y .

We now present some examples of the function `randvar_getvalue ( rv , rv2 , value2 )`, which performs this transformation. The statement

```
value = randvar_getvalue ( rv , rv2 , value2 )
```

returns a random value from the distribution function of the random variable `rv` by transformation of `value2` from the distribution function of random variable `rv2`.

In the following session, we transform a uniform random variable sample into a LogUniform variable sample. We begin to create a random variable `rv` from a LogUniform law and parameters $a = 10, b = 20$. Then we create a second random variable `rv2` from a Uniforme law and parameters $a = 2, b = 3$. The main loop is based on the transformation of a sample computed from `rv2` into a sample from `rv`. The `mean` allows to check that the transformed samples have an mean value which corresponds to the random variable `rv`.

```
nisp_initseed ( 0 );
a = 10.0;
b = 20.0;
rv = randvar_new ( "LogUniforme" , a , b );
rv2 = randvar_new ( "Uniforme" , 2 , 3 );
nbshots = 1000;
valuesLou = zeros(nbshots);
for i=1:nbshots
    valuesUni(i) = randvar_getvalue( rv2 );
    valuesLou(i) = randvar_getvalue( rv , rv2 , valuesUni(i) );
end
computed = mean (valuesLou);
mu = (b-a)/(log(b)-log(a));
expected = mu;
mprintf("Expectation=%.5f_(expected=%.5f)\n",computed,expected);
//
scf();
histplot(50,valuesUni);
xtitle("Empirical_histogram_-_Uniform_variable","X","P(X)");
scf();
histplot(50,valuesLou);
xtitle("Empirical_histogram_-_Log-Uniform_variable","X","P(X)");
randvar_destroy(rv);
randvar_destroy(rv2);
```

The previous script produces the following output.

```
Expectation=14.63075 (expected=14.42695)
```

The previous script also produces the figures [2.5](#) and [2.6](#).

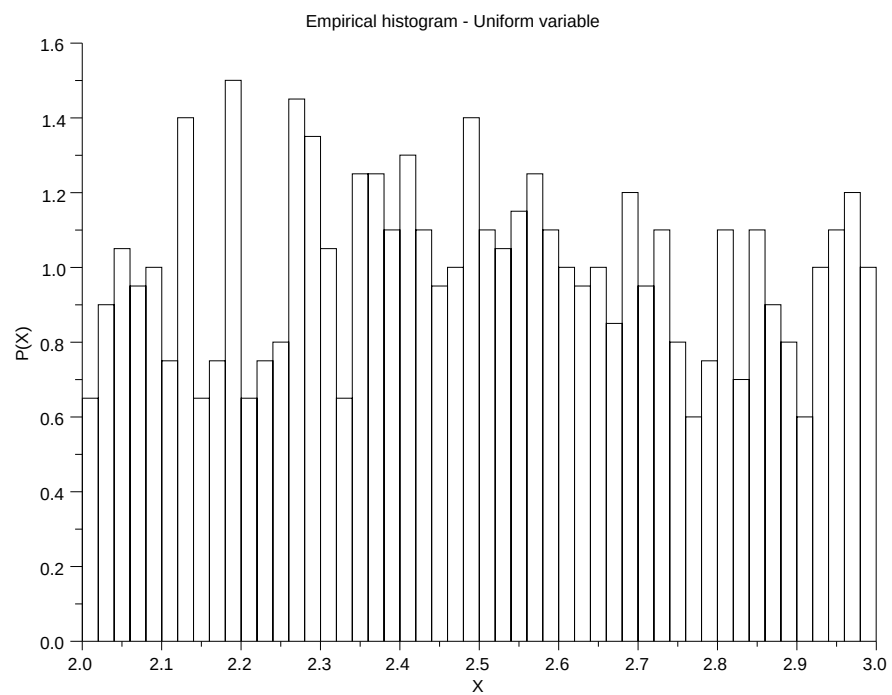


Figure 2.5: The histogram of a Uniform random variable with 1000 samples.

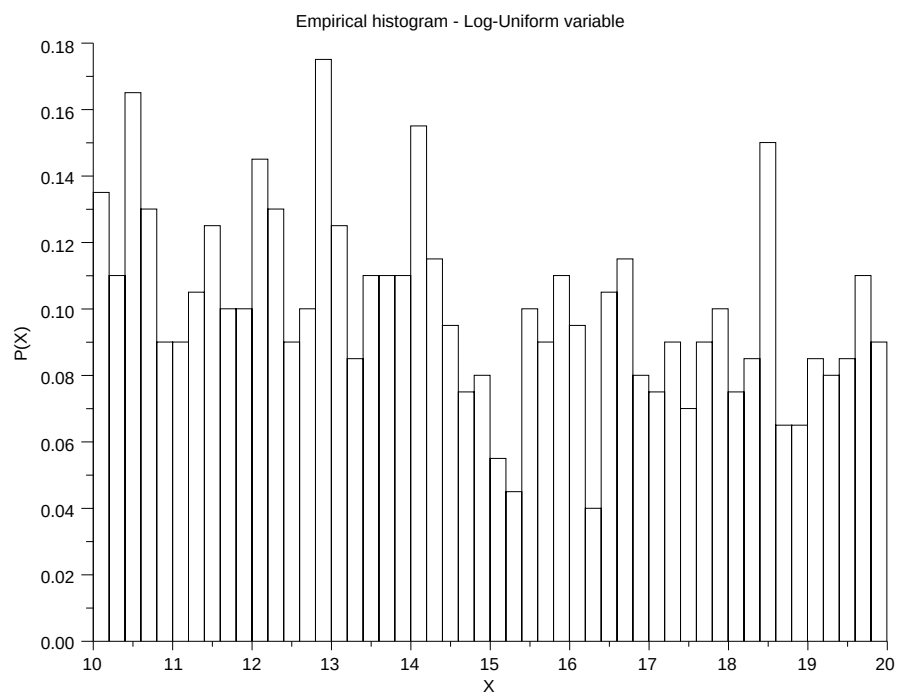


Figure 2.6: The histogram of a Log-Uniform random variable with 1000 samples.

Chapter 3

The `setrandvar` class

In this chapter, we present the `setrandvar` class. The first section gives a brief outline of the features of this class and the second section presents several examples.

3.1 Introduction

The `setrandvar` class allows to manage a collection of random variables and to build a Design Of Experiments (DOE). Several types of DOE are provided:

- Monte-Carlo,
- Latin Hypercube Sampling,
- Smolyak.

Once a DOE is created, we can retrieve the information experiment by experiment or the whole matrix of experiments. This last feature allows to benefit from the fact that Scilab can natively manage matrices, so that we do not have to perform loops to manage the complete DOE. Hence, good performances can be observed, even if the language still is interpreted.

The figure [3.1](#) presents the methods available in the `setrandvar` class. A complete description of the input and output arguments of each function is available in the inline help and will not be repeated here.

More informations about the Oriented Object system used in this toolbox can be found in the section [2.2.2](#).

3.2 Examples

In this section, we present examples of use of the `setrandvar` class. In the first example, we present a Scilab session where we create a Latin Hypercube Sampling. In the second part, we present various types of DOE which can be generated with this class.

3.2.1 A Monte-Carlo design with 2 variables

In the following example, we build a Monte-Carlo design of experiments, with 2 input random variables. The first variable is associated with a Normal distribution function and the second

Constructors <code>srv = setrandvar_new ()</code> <code>srv = setrandvar_new (n)</code> <code>srv = setrandvar_new (file)</code>
Methods <code>setrandvar_setsample (srv , name , np)</code> <code>setrandvar_setsample (srv , k , i , value)</code> <code>setrandvar_setsample (srv , k , value)</code> <code>setrandvar_setsample (srv , value)</code> <code>setrandvar_save (srv , file)</code> <code>np = setrandvar_getsize (srv)</code> <code>sample = setrandvar_getsample (srv , k , i)</code> <code>sample = setrandvar_getsample (srv , k)</code> <code>sample = setrandvar_getsample (srv)</code> <code>setrandvar_getlog (srv)</code> <code>nx = setrandvar_getdimension (srv)</code> <code>setrandvar_freememory (srv)</code> <code>setrandvar_buildsample (srv , srv2)</code> <code>setrandvar_buildsample (srv , name , np)</code> <code>setrandvar_buildsample (srv , name , np , ne)</code> <code>setrandvar_addrandvar (srv , rv)</code>
Destructor <code>setrandvar_destroy (srv)</code>
Static methods <code>tokenmatrix = setrandvar_tokens ()</code> <code>nb = setrandvar_size ()</code>

Figure 3.1: Outline of the methods of the `setrandvar` class

variable is associated with a Uniform distribution function. The simulation is based on 1000 experiments.

The function `nisp_initseed` is used to set the value of the seed to zero, so that the results can be reproduced. The `setrandvar_new` function is used to create a new set of random variables. Then we create two new random variables with the `randvar_new` function. These two variables are added to the set with the `setrandvar_addrandvar` function. The `setrandvar_buildsample` allows to build the design of experiments, which can be retrieved as matrix with the `setrandvar_getsample` function. The sampling matrix has `np` rows and 2 columns (one for each input variable).

```
nisp_initseed(0);
rvu1 = randvar_new("Normale",1,3);
rvu2 = randvar_new("Uniforme",2,3);
//
srvu = setrandvar_new();
setrandvar_addrandvar ( srvu, rvu1);
setrandvar_addrandvar ( srvu, rvu2);
//
np = 5000;
setrandvar_buildsample(srvu, "MonteCarlo",np);
sampling = setrandvar_getsample(srvu);
// Check sampling of random variable #1
mean(sampling(:,1)) // Expectation : 1
// Check sampling of random variable #2
mean(sampling(:,2)) // Expectation : 2.5
//
scf();
histplot(50,sampling(:,1));
xtitle("Empirical_histogram_of_X1");
scf();
histplot(50,sampling(:,2));
xtitle("Empirical_histogram_of_X2");
//
// Clean-up
setrandvar_destroy(srvu);
randvar_destroy(rvu1);
randvar_destroy(rvu2);
```

The previous script produces the following output.

```
-->mean(sampling(:,1)) // Expectation : 1
ans =
    1.0064346
-->mean(sampling(:,2)) // Expectation : 2.5
ans =
    2.5030984
```

The previous script also produces the figures [3.2](#) and [3.3](#).

We may now want to add the exact distribution to these histograms and compare. The Normal distribution function is not provided by Scilab, but is provided by the `Distfun` module. Indeed, the `distfun_normpdf` function of the `Distfun` module computes the Normal probability

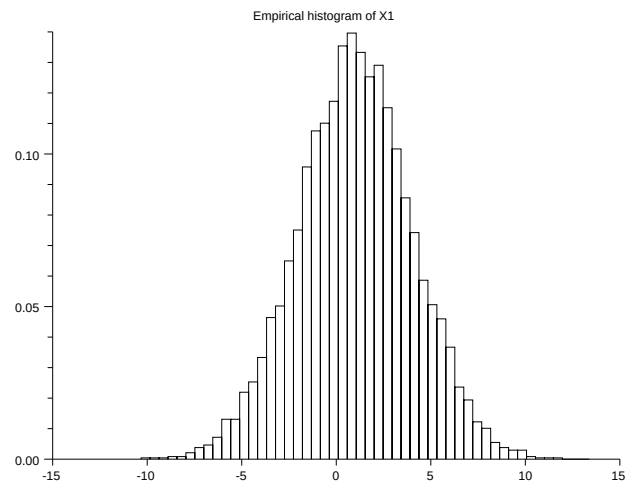


Figure 3.2: Monte-Carlo Sampling - Normal random variable.

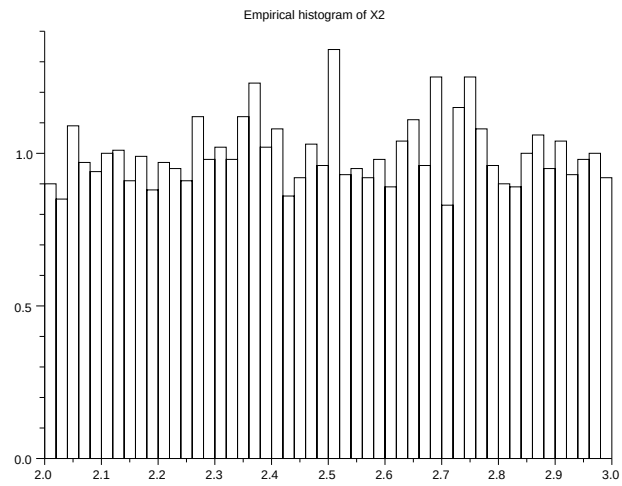


Figure 3.3: Monte-Carlo Sampling - Uniform random variable.

distribution function. In order to install this module, we can run the `atomsInstall` function, as in the following script.

```
atomsInstall("distfun")
```

The following script compares the empirical and theoretical distributions.

```
scf();
histplot(50,sampling(:,1));
xtitle("Empirical_histogram_of_X1");
x=linspace(-15,15,1000);
y = dnorm(x,1,3);
plot(x,y,"r-")
legend(["Empirical","Exact"]);
```

The previous script produces the figure 3.4.

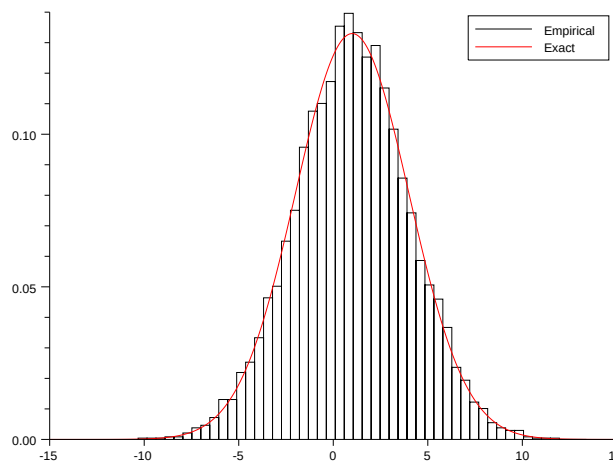


Figure 3.4: Monte-Carlo Sampling - Histogram and exact distribution functions for the first variable.

The following script performs the same comparison for the second variable.

```
scf();
histplot(50,sampling(:,2));
xtitle("Empirical_histogram_of_X2");
x=linspace(2,3,1000);
y=ones(1000,1);
plot(x,y,"r-");
```

The previous script produces the figure 3.5.

3.2.2 A Monte-Carlo design with 2 variables

In this section, we create a Monte-Carlo design with 2 variables.



Figure 3.5: Monte-Carlo Sampling - Histogram and exact distribution functions for the second variable.

We are going to use the exponential distribution function, which is not defined in Scilab. The following `exppdf` function computes the probability distribution function of the exponential distribution function.

```
function p = exppdf ( x , lambda )
    p = lambda.*exp(-lambda.*x)
endfunction
```

The following script creates a Monte-Carlo sampling where the first variable is Normal and the second variable is Exponential. Then we compare the empirical histogram and the exact distribution function. We use the `dnorm` function defined in the Stixbox module.

```
nisp_initseed ( 0 );
rv1 = randvar_new("Normale",1.0,0.5);
rv2 = randvar_new("Exponentielle",5.);
// Definition d'un groupe de variables aleatoires
srv = setrandvar_new ( );
setrandvar_addrandvar ( srv , rv1 );
setrandvar_addrandvar ( srv , rv2 );
np = 1000;
setrandvar_buildsample ( srv , "MonteCarlo" , np );
//
sampling = setrandvar_getsample ( srv );
// Check sampling of random variable #1
mean(sampling(:,1)), variance(sampling(:,1))
// Check sampling of random variable #2
min(sampling(:,2)), max(sampling(:,2))
// Plot
scf();
histplot(40, sampling(:,1))
```

```

x = linspace(-1,3,1000)';
p = dnorm(x,1,0.5);
plot(x,p,"r-")
xlabel("Empirical histogram of X1","X","P(X)");
legend(["Empirical","Exact"]);
scf();
histplot(40, sampling(:,2))
x = linspace(0,2,1000)';
p = exppdf ( x , 5 );
plot(x,p,"r-")
xlabel("Empirical histogram of X2","X","P(X)");
legend(["Empirical","Exact"]);
// Clean-up
setrandvar_destroy(srv);
randvar_destroy(rv1);
randvar_destroy(rv2);

```

The previous script produces the figures 3.6 and 3.7.

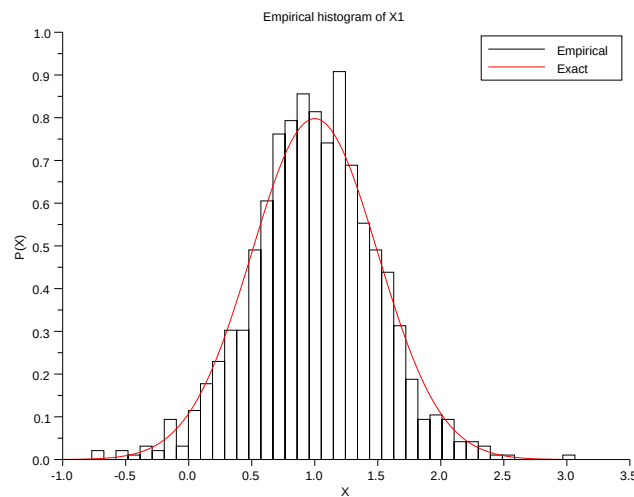


Figure 3.6: Monte-Carlo Sampling - Histogram and exact distribution functions for the first variable.

3.2.3 A LHS design

In this section, we present the creation of a Latin Hypercube Sampling. In our example, the DOE is based on two random variables, the first being Normal with mean 1.0 and standard deviation 0.5 and the second being Uniform in the interval [2, 3].

We begin by defining two random variables with the `randvar_new` function.

```

vu1 = randvar_new("Normale",1.0,0.5);
vu2 = randvar_new("Uniforme",2.0,3.0);

```

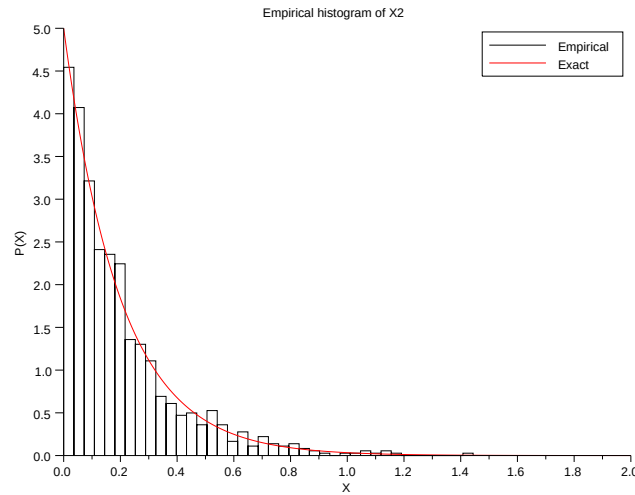


Figure 3.7: Monte-Carlo Sampling - Histogram and exact distribution functions for the second variable.

Then, we create a collection of random variables with the `setrandvar_new` function which creates here an empty collection of random variables. Then we add the two random variables to the collection.

```
srv = setrandvar_new ( );
setrandvar_addrandvar ( srv , vu1 );
setrandvar_addrandvar ( srv , vu2 );
```

We can now build the DOE so that it is a LHS sampling with 1000 experiments.

```
setrandvar_buildsample ( srv , "Lhs" , 1000 );
```

At this point, the DOE is stored in the memory space of the NISP library, but we do not have a direct access to it. We now call the `setrandvar_getsample` function and store that DOE into the `sampling` matrix.

```
sampling = setrandvar_getsample ( srv );
```

The `sampling` matrix has 1000 rows, corresponding to each experiment, and 2 columns, corresponding to each input random variable.

The following script allows to plot the sampling, which is presented in figure 3.8.

```
my_handle = scf();
clf(my_handle,"reset");
plot(sampling(:,1),sampling(:,2));
my_handle.children.children.children.line_mode = "off";
my_handle.children.children.children.mark_mode = "on";
my_handle.children.children.children.mark_size = 2;
my_handle.children.title.text = "Latin_Hypercube_Sampling";
my_handle.children.x_label.text = "Variable_#1:_Normale,1.0,0.5";
my_handle.children.y_label.text = "Variable_#2:_Uniforme,2.0,3.0";
```

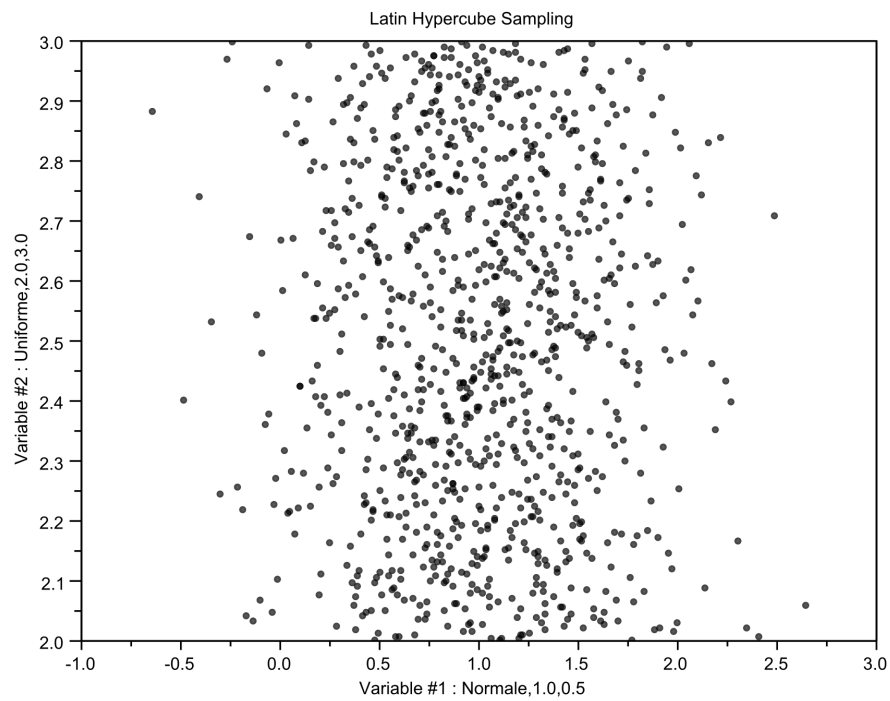


Figure 3.8: Latin Hypercube Sampling - The first variable is Normal, the second variable is Uniform.

The following script allows to plot the histogram of the two variables, which are presented in figures 3.9 and 3.10.

```
// Plot Var #1
my_handle = scf();
clf(my_handle,"reset");
histplot ( 50 , sampling(:,1))
my_handle.children.title.text = "Variable_#1:_Normale,1.0,0.5";
// Plot Var #2
my_handle = scf();
clf(my_handle,"reset");
histplot ( 50 , sampling(:,2))
my_handle.children.title.text = "Variable_#2:_Uniforme,2.0,3.0";
```

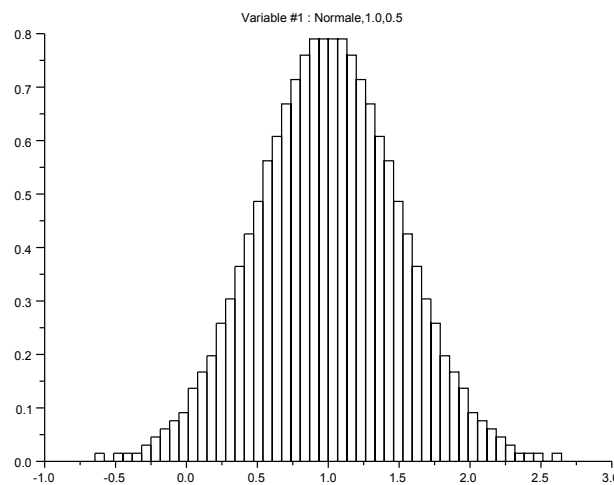


Figure 3.9: Latin Hypercube Sampling - Normal random variable.

We can use the **mean** and **variance** on each random variable and check that the expected result is computed. We insist on the fact that the **mean** and **variance** functions are not provided by the NISP library: these are pre-defined functions which are available in the Scilab library. That means that any Scilab function can be now used to process the data generated by the toolbox.

```
for ivar = 1:2
    m = mean(sampling(:,ivar))
    mprintf("Variable_#%d,_Mean:_%f\n",ivar,m)
    v = variance(sampling(:,ivar))
    mprintf("Variable_#%d,_Variance:_%f\n",ivar,v)
end
```

The previous script produces the following output.

```
Variable #1, Mean : 1.000000
Variable #1, Variance : 0.249925
Variable #2, Mean : 2.500000
Variable #2, Variance : 0.083417
```

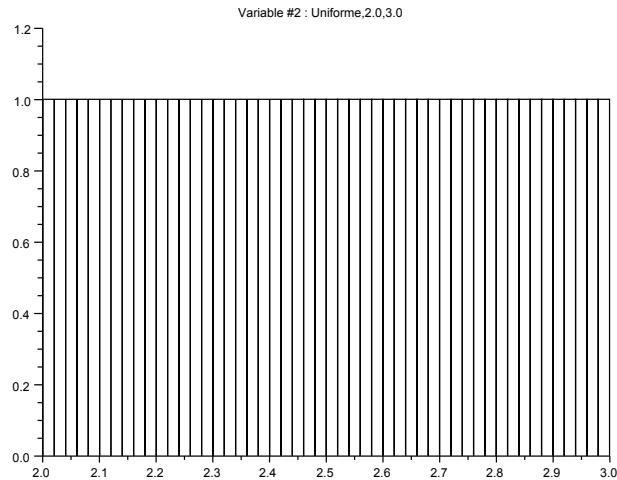


Figure 3.10: Latin Hypercube Sampling - Uniform random variable.

Our numerical simulation is now finished, but we must destroy the objects so that the memory managed by the toolbox is deleted.

```
randvar_destroy(vu1)
randvar_destroy(vu2)
setrandvar_destroy(srv)
```

3.2.4 A note on the LHS samplings

We emphasize that the LHS sampling which is provided by the `setrandvar_buildsample` function is so that the points are centered within their cells.

In the following script, we create a LHS sampling with 10 points.

```
srv = setrandvar_new(2);
np = 10;
setrandvar_buildsample ( srv , "Lhs" , np );
sampling = setrandvar_getsample ( srv );
scf();
plot(sampling(:,1),sampling(:,2),"bo");
xtitle("LHS□Design","X1","X2");
// Add the cuts
cut = linspace ( 0 , 1 , np + 1 );
for i = 1 : np + 1
    plot( [cut(i) cut(i)] , [0 1] , "-" )
end
for i = 1 : np + 1
    plot( [0 1] , [cut(i) cut(i)] , "-" )
end
setrandvar_destroy ( srv )
```

The previous script produces the figure 3.11.

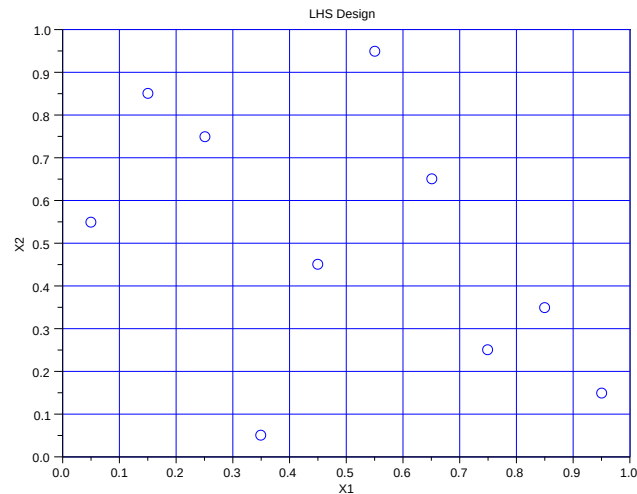


Figure 3.11: Latin Hypercube Sampling - Computed with `setrandvar_buildsample` and the "Lhs" option.

The "LhsMaxMin" sampling provided by the `setrandvar_buildsample` function tries to maximize the minimum distance between the points in the sampling. The `ntry` parameter is the number of random points generated before the best is accepted in the sampling.

```
np = 10;
ntry = 100;
setrandvar_buildsample ( srv , "LhsMaxMin" , np , ntry );
sampling = setrandvar_getsample ( srv );
```

The previous script produces the figure 3.12.

On the other hand, the `nisp_buildlhs` function produces a more classical LHS sampling, where the points are randomly picked within their cells.

```
n = 5;
s = 2;
sampling = nisp_buildlhs ( s , n );
scf();
plot ( sampling(:,1) , sampling(:,2) , "bo" );
// Add the cuts
cut = linspace ( 0 , 1 , n + 1 );
for i = 1 : n + 1
plot( [cut(i) cut(i)] , [0 1] , "-" )
end
for i = 1 : n + 1
plot( [0 1] , [cut(i) cut(i)] , "-" )
end
```

The previous script produces the figure 3.13.

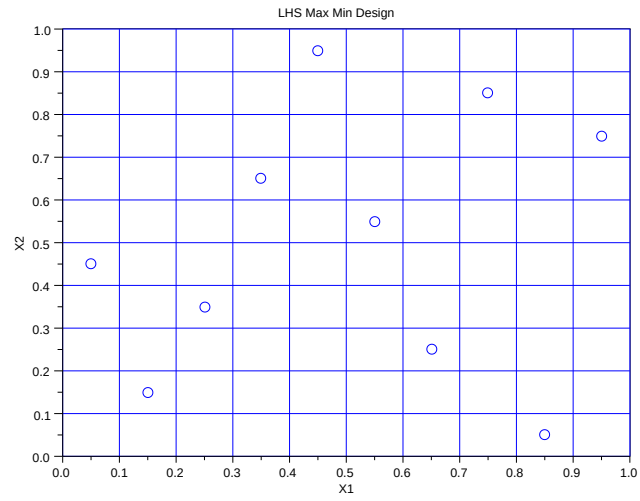


Figure 3.12: Latin Hypercube Sampling - Computed with `setrandvar_buildsample` and the "LhsMaxMin" option.

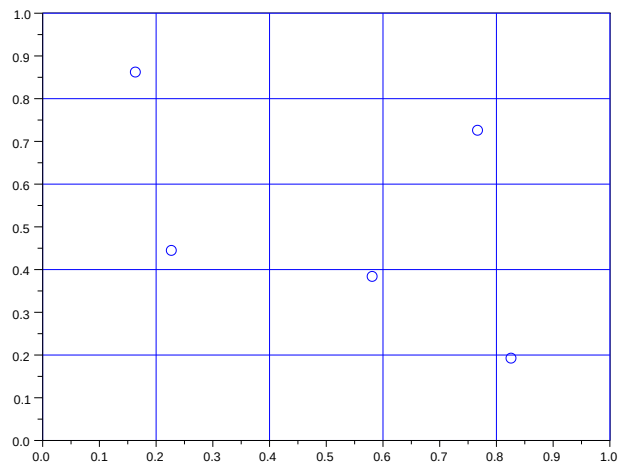


Figure 3.13: Latin Hypercube Sampling - Computed with `nisp_buildlhs`.

3.2.5 Other types of DOEs

The following Scilab session allows to generate a Monte-Carlo sampling with two uniform variables in the interval $[-1, 1]$. The figure 3.14 presents this sampling and the figures 3.15 and 3.16 present the histograms of the two uniform random variables.

```
vu1 = randvar_new("Uniforme",-1.0,1.0);
vu2 = randvar_new("Uniforme",-1.0,1.0);
srv = setrandvar_new ( );
setrandvar_addrandvar ( srv , vu1 );
setrandvar_addrandvar ( srv , vu2 );
setrandvar_buildsample ( srv , "MonteCarlo" , 1000 );
sampling = setrandvar_getsample ( srv );
randvar_destroy(vu1);
randvar_destroy(vu2);
setrandvar_destroy(srv);
```

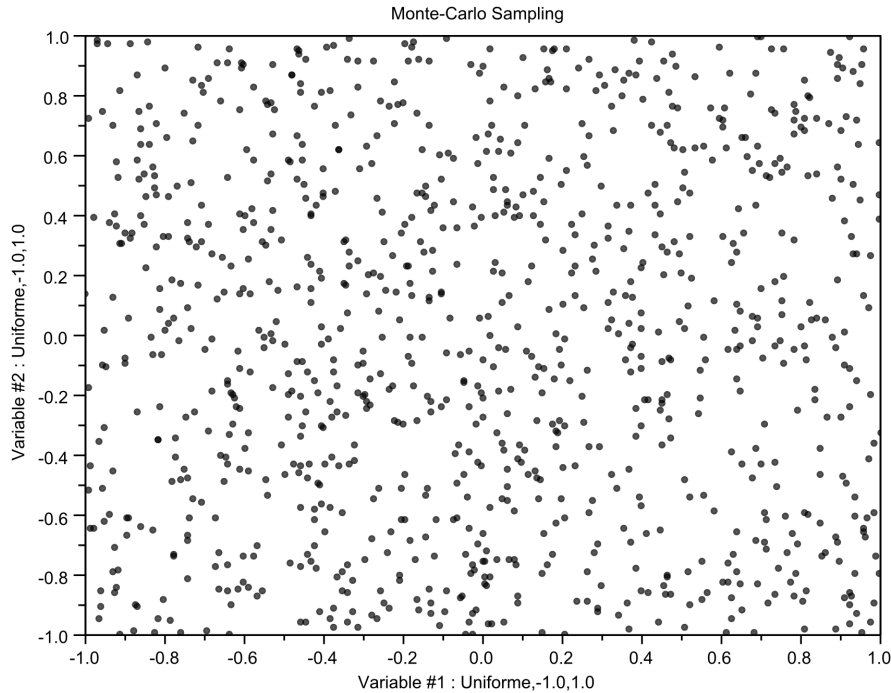


Figure 3.14: Monte-Carlo Sampling - Two uniform variables in the interval $[-1, 1]$.

With the `setrandvar_buildsample` function, we can change the type of sampling by changing the second argument. This way, we can create the Petras, Quadrature and Sobol sampling presented in figures 3.17, 3.18 and 3.19.

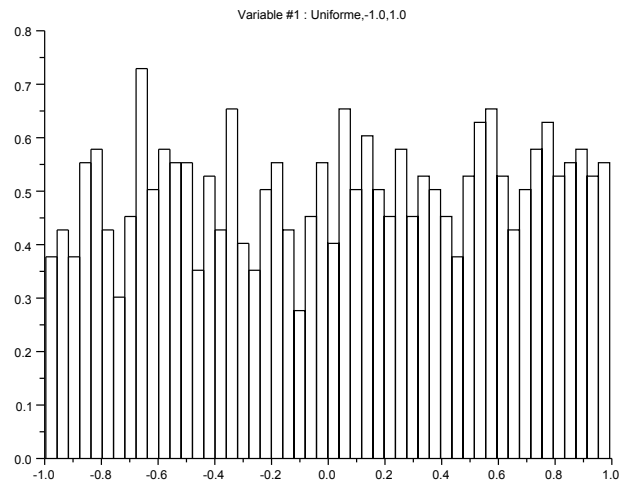


Figure 3.15: Latin Hypercube Sampling - First uniform variable in $[-1, 1]$.

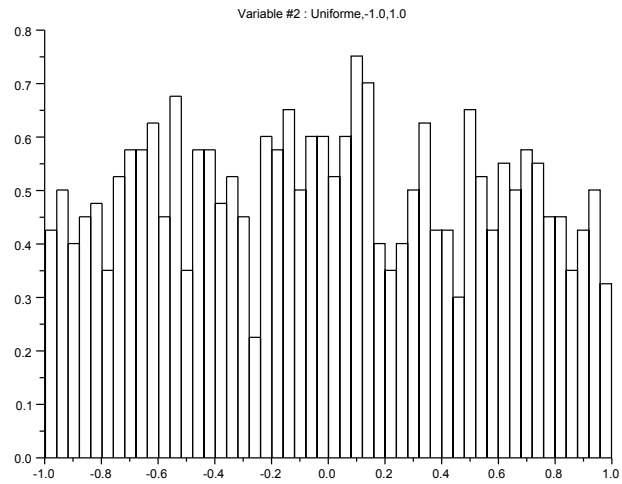


Figure 3.16: Latin Hypercube Sampling - Second uniform variable in $[-1, 1]$.

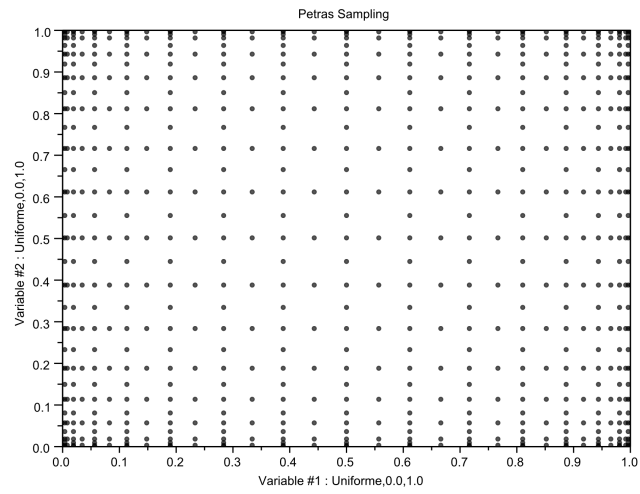


Figure 3.17: Petras sampling - Two uniform variables in the interval $[-1, 1]$.

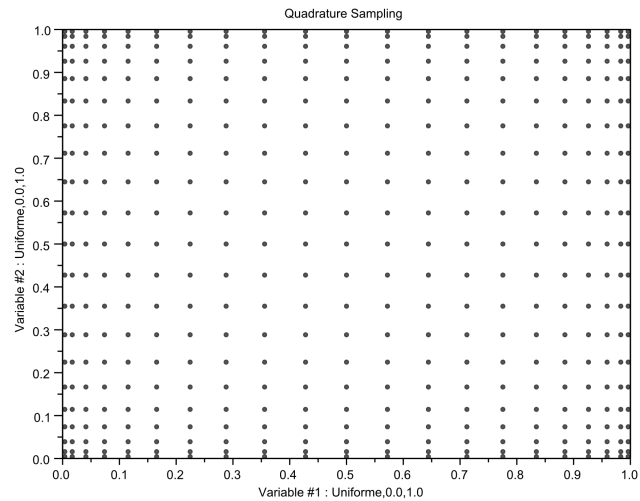


Figure 3.18: Quadrature sampling - Two uniform variables in the interval $[-1, 1]$.

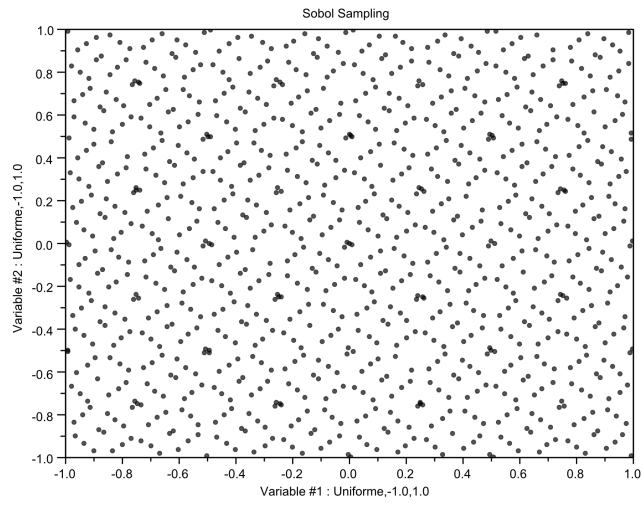


Figure 3.19: Sobol sampling - Two uniform variables in the interval $[-1, 1]$.

Chapter 4

The polychaos class

4.1 Introduction

The `polychaos` class allows to manage a polynomial chaos expansion. The coefficients of the expansion are computed based on given numerical experiments which creates the association between the inputs and the outputs. Once computed, the expansion can be used as a regular function. The mean, standard deviation or quantile can also be directly retrieved.

The tool allows to get the following results:

- mean,
- variance,
- quantile,
- correlation, etc...

Moreover, we can generate the C source code which computes the output of the polynomial chaos expansion. This C source code is stand-alone, that is, it is independent of both the NISP library and Scilab. It can be used as a meta-model.

The figure [4.1](#) presents the most commonly used methods available in the `polychaos` class. More methods are presented in figure [4.2](#). The inline help contains the detailed calling sequence for each function and will not be repeated here. More than 50 methods are available and most of them will not be presented here.

More informations about the Oriented Object system used in this toolbox can be found in the section [2.2.2](#).

4.2 Examples

In this section, we present to examples of use of the `polychaos` class.

4.2.1 Product of two random variables

In this section, we present the polynomial expansion of the product of two random variables. We analyse the Scilab script and present the methods which are available to perform the sensi-

Constructors
<code>pc = polychaos_new (file)</code>
<code>pc = polychaos_new (srv , ny)</code>
<code>pc = polychaos_new (pc , nopt , varopt)</code>
Methods
<code>polychaos_setsizetarget (pc , np)</code>
<code>polychaos_settarget (pc , output)</code>
<code>polychaos_setinput (pc , invalue)</code>
<code>polychaos_setdimoutput (pc , ny)</code>
<code>polychaos_setdegree (pc , no)</code>
<code>polychaos_getvariance (pc)</code>
<code>polychaos_getmean (pc)</code>
Destructor
<code>polychaos_destroy (pc)</code>
Static methods
<code>tokenmatrix = polychaos_tokens ()</code>
<code>nb = polychaos_size ()</code>

Figure 4.1: Outline of the methods of the `polychaos` class

tivity analysis. This script is based on the NISP methodology, which has been presented in the Introduction chapter. We will use the figure 1.1 as a framework and will follow the steps in order.

In the following Scilab script, we define the function `Exemple` which takes a vector of size 2 as input and returns a scalar as output.

```
function y = Exemple (x)
    y(:,1) = x(:,1) .* x(:,2)
endfunction
```

We now create a collection of two stochastic (normalized) random variables. Since the random variables are normalized, we use the default parameters of the `randvar_new` function. The normalized collection is stored in the variable `srvx`.

```
vx1 = randvar_new("Normale");
vx2 = randvar_new("Uniforme");
srvx = setrandvar_new();
setrandvar_addrandvar ( srvx , vx1 );
setrandvar_addrandvar ( srvx , vx2 );
```

We create a collection of two uncertain parameters. We explicitly set the parameters of each random variable, that is, the first Normal variable is associated with a mean equal to 1.0 and a standard deviation equal to 0.5, while the second Uniform variable is in the interval [1.0, 2.5]. This collection is stored in the variable `srvu`.

```
vu1 = randvar_new("Normale",1.0,0.5);
vu2 = randvar_new("Uniforme",1.0,2.5);
srvu = setrandvar_new();
setrandvar_addrandvar ( srvu , vu1 );
setrandvar_addrandvar ( srvu , vu2 );
```

Methods

```
output = polychaos_gettarget ( pc )
np = polychaos_getsizetarget ( pc )
polychaos_getsample ( pc , k , ovar )
polychaos_getquantile ( pc , k )
polychaos_getsample ( pc )
polychaos_getquantile ( pc , alpha )
polychaos_getoutput ( pc )
polychaos_getmultind ( pc )
polychaos_getlog ( pc )
polychaos_getinvquantile ( pc , threshold )
polychaos_getindextotal ( pc )
polychaos_getindexfirst ( pc )
ny = polychaos_getdimoutput ( pc )
nx = polychaos_getdiminput ( pc )
p = polychaos_getdimexp ( pc )
no = polychaos_getdegree ( pc )
polychaos_getcovariance ( pc )
polychaos_getcorrelation ( pc )
polychaos_getanova ( pc )
polychaos_generatecode ( pc , filename , funname )
polychaos_computeoutput ( pc )
polychaos_computeexp ( pc , srv , method )
polychaos_computeexp ( pc , pc2 , invalue , varopt )
polychaos_buildsample ( pc , type , np , order )
```

Figure 4.2: More methods from the `polychaos` class

The first design of experiment is build on the stochastic set `srvx` and based on a Quadrature type of DOE. Then this DOE is transformed into a DOE for the uncertain collection of parameters `srvu`.

```
degree = 2;
setrandvar_buildsample ( srvx , "Quadrature" , degree );
setrandvar_buildsample ( srvu , srvx );
```

The next steps will be to create the polynomial and actually perform the DOE. But before doing this, we can take a look at the DOE associated with the stochastic and uncertain collection of random variables. We can use the `setrandvar_getsample` twice and get the following output.

```
-->setrandvar_getsample(srvx)
ans =

- 1.7320508    0.1127017
- 1.7320508    0.5
- 1.7320508    0.8872983
  0.          0.1127017
  0.          0.5
  0.          0.8872983
  1.7320508    0.1127017
  1.7320508    0.5
  1.7320508    0.8872983
-->setrandvar_getsample(srvu)
ans =

  0.1339746    1.1690525
  0.1339746    1.75
  0.1339746    2.3309475
  1.          1.1690525
  1.          1.75
  1.          2.3309475
  1.8660254    1.1690525
  1.8660254    1.75
  1.8660254    2.3309475
```

These two matrices are a 9×2 matrices, where each line represents an experiment and each column represents an input random variable. The stochastic (normalized) `srvx` DOE has been created first, then the `srvu` has been deduced from `srvx` based on random variable transformations.

We now use the `polychaos_new` function and create a new polynomial `pc`. The number of input variables corresponds to the number of variables in the stochastic collection `srvx`, that is 2, and the number of output variables is given as the input argument `ny`. In this particular case, the number of experiments to perform is equal to `np=9`, as returned by the `setrandvar_getsize` function. This parameter is passed to the polynomial `pc` with the `polychaos_setsizetarget` function.

```
ny = 1;
pc = polychaos_new ( srvx , ny );
np = setrandvar_getsize(srvx);
polychaos_setsizetarget(pc,np);
```

In the next step, we perform the simulations prescribed by the DOE. We perform this loop in the Scilab language and make a loop over the index `k`, which represents the index of the current experiment, while `np` is the total number of experiments to perform. For each loop, we get the input from the uncertain collection `srvu` with the `setrandvar_getsample` function, pass it to the `Exemple` function, get back the output which is then transferred to the polynomial `pc` by the `polychaos_settarget` function.

```
inputdata = setrandvar_getsample(srvu);
outputdata = Exemple(inputdata);
polychaos_settarget(pc, outputdata);
```

We can compute the polynomial expansion based on numerical integration so that the coefficients of the polynomial are determined. This is done with the `polychaos_computeexp` function, which stands for "compute the expansion".

```
polychaos_setdegree(pc, degree);
polychaos_computeexp(pc, srvx, "Integration");
```

Everything is now ready for the sensitivity analysis. Indeed, the `polychaos_getmean` returns the mean while the `polychaos_getvariance` returns the variance.

```
average = polychaos_getmean(pc);
var = polychaos_getvariance(pc);
mprintf("Mean_=====%f\n", average);
mprintf("Variance_=====%f\n", var);
mprintf("Indice_de_sensibilite_du_1er_ordre\n");
mprintf("====Variable_X1_=%f\n", polychaos_getindexfirst(pc, 1));
mprintf("====Variable_X2_=%f\n", polychaos_getindexfirst(pc, 2));
mprintf("Indice_de_sensibilite_Totale\n");
mprintf("====Variable_X1_=%f\n", polychaos_getindextotal(pc, 1));
mprintf("====Variable_X2_=%f\n", polychaos_getindextotal(pc, 2));
```

The previous script produces the following output.

```
Mean      = 1.750000
Variance   = 1.000000
Indice de sensibilite du 1er ordre
    Variable X1 = 0.765625
    Variable X2 = 0.187500
Indice de sensibilite Totale
    Variable X1 = 0.812500
    Variable X2 = 0.234375
```

In order to free the memory required for the computation, it is necessary to delete all the objects created so far.

```
polychaos_destroy(pc);
randvar_destroy(vu1);
randvar_destroy(vu2);
randvar_destroy(vx1);
randvar_destroy(vx2);
setrandvar_destroy(srvu);
setrandvar_destroy(srvx);
```

Prior to destroying the objects, we can inquire a little more about the density of the output of the chaos polynomial. In the following script, we create a Latin Hypercube Sampling made of 10 000 points. Then get the output of the polynomial on these inputs and plot the histogram of the output.

```
polychaos_buildsample(pc,"Lhs",10000,0);
sample_output = polychaos_getsample(pc);
scf();
histplot(50,sample_output);
xtitle("Product function - Empirical Histogram","X","P(X)");
```

The previous script produces the figure 4.3.

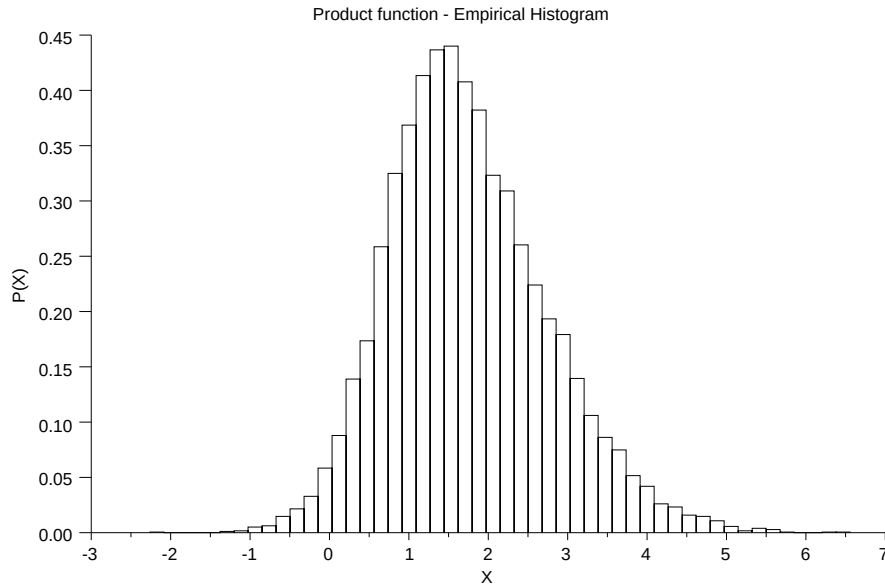


Figure 4.3: Product function - Histogram of the output on a LHS design with 10000 experiments.

We may explore the following topics.

- Perform the same analysis where the variable X_2 is a normal variable with mean 2 and standard deviation 2.
- Check that the development in polynomial chaos on a Hermite-Hermite basis does not allow to get exact results. See that the convergence can be obtained by increasing the degree.
- Check that the development on a basis Hermite-Legendre allows to get exact results with degree 2.

4.2.2 A note on performance

In this section, we emphasize vectorization which can be used to improve the performance of a script when we compute the output of a function on a given sampling.

In order to use vectorization, the core feature that we used in the `Exemple` is the use of the elementwise multiplication, denoted by `.*`. In the `Exemple` function below (reproduced here for simplicity), the input `x` is a `np-by-2` matrix of doubles, where `np` is the number of experiments, and `y` is a `np-by-1` matrix of doubles.

```
function y = Exemple (x)
    y(:,1) = x(:,1) .* x(:,2)
endfunction
```

The elementwise multiplication allows to multiply the two first columns of `x`, and sets the result into the output `y`, in one single statement. Since Scilab uses optimized numerical libraries, this allows to get the best performance in most situations.

In the previous section, we have shown that we can compute the output of the `Exemple` function in one single call to the function.

```
outputdata = Exemple(inputdata);
```

This call allows to produce all the outputs as fast as possible and is the recommended method. The reason is that the previous script lets Scilab perform computations with large matrices.

In fact, there is another, slower, method to perform the same computation. We make a loop over the index `k`, which represents the index of the current experiment, while `np` is the total number of experiments to perform. For each loop, we get the input from the uncertain collection `srvu` with the `setrandvar_getsample` function, pass it to the `Exemple` function, get back the output which is then transferred to the polynomial `pc` by the `polychaos_settarget` function.

```
// This is slow.
for k=1:np
    inputdata = setrandvar_getsample(srvu,k);
    outputdata = Exemple(inputdata);
    mprintf ( "Experiment %d, input=[%f %f], output=%f\n", k, ..
        inputdata(1), inputdata(2) , outputdata )
    polychaos_settarget(pc,k,outputdata);
end
```

The previous script produces the following output.

```
Experiment #1, input =[0.133975 1.169052], output = 0.156623
Experiment #2, input =[0.133975 1.750000], output = 0.234456
Experiment #3, input =[0.133975 2.330948], output = 0.312288
Experiment #4, input =[1.000000 1.169052], output = 1.169052
Experiment #5, input =[1.000000 1.750000], output = 1.750000
Experiment #6, input =[1.000000 2.330948], output = 2.330948
Experiment #7, input =[1.866025 1.169052], output = 2.181482
Experiment #8, input =[1.866025 1.750000], output = 3.265544
Experiment #9, input =[1.866025 2.330948], output = 4.349607
```

While the previous script is perfectly correct, it can be very slow when the number of experiments is large. This is because the interpreter has to perform a large number of loops with matrices of small size. In general, this produces much slower script and should be avoided. More details on this topic are presented in [1].

4.2.3 The Ishigami test case

In this section, we present the Ishigami test case.

The function `Exemple` is the model that we consider in this numerical experiment. This function takes a vector of size 3 in input and returns a scalar output.

```
function y = Exemple (x)
    a=7.
    b=0.1
    s1=sin(x(:,1))
    s2=sin(x(:,2))
    y(:,1) = s1 + a.*s2.*s2 + b.*x(:,3).*x(:,3).*x(:,3).*x(:,3).*s1
endfunction
```

We create 3 uncertain parameters which are uniform in the interval $[-\pi, \pi]$ and put these random variables into the collection `srvu`.

```
rvu1 = randvar_new("Uniforme",-%pi,%pi);
rvu2 = randvar_new("Uniforme",-%pi,%pi);
rvu3 = randvar_new("Uniforme",-%pi,%pi);

srvu = setrandvar_new();
setrandvar_addrandvar ( srvu, rvu1);
setrandvar_addrandvar ( srvu, rvu2);
setrandvar_addrandvar ( srvu, rvu3);
```

The collection of stochastic variables is created with the function `setrandvar_new`. The calling sequence `srvx = setrandvar_new( nx )` allows to create a collection of `nx=3` random variables uniform in the interval $[0, 1]$. Then we create a Petras DOE for the stochastic collection `srvx` and transform it into a DOE for the uncertain parameters `srvu`.

```
nx = setrandvar_getdimension ( srvu );
srvx = setrandvar_new( nx );
degre = 9;
setrandvar_buildsample(srvx,"Petras",degre);
setrandvar_buildsample( srvu , srvx );
```

We use the `polychaos_new` function and create the new polynomial `pc` with 3 inputs and 1 output.

```
noutput = 1;
pc = polychaos_new ( srvx , noutput );
```

The next step allows to perform the simulations associated with the DOE prescribed by the collection `srvu`. Here, we must perform `np=751` experiments.

```
np = setrandvar_getsize(srvu);
polychaos_setsizetarget(pc,np);
inputdata = setrandvar_getsample(srvu);
outputdata = Exemple(inputdata);
polychaos_settarget(pc,outputdata);
```

We can now compute the polynomial expansion by integration.

```
polychaos_setdegree(pc,degre);
polychaos_computeexp(pc,srvx,"Integration");
```

Everything is now ready so that we can do the sensitivity analysis, as in the following script.

```
average = polychaos_getmean(pc);
var = polychaos_getvariance(pc);
mprintf("Mean░░░░░░░░░░=░%f\n",average);
mprintf("Variance░░░░░=░%f\n",var);
mprintf("First░order░sensitivity░index\n");
mprintf("░░░░Variable░X1░=░%f\n",polychaos_getindexfirst(pc,1));
mprintf("░░░░Variable░X2░=░%f\n",polychaos_getindexfirst(pc,2));
mprintf("░░░░Variable░X3░=░%f\n",polychaos_getindexfirst(pc,3));
mprintf("Total░sensitivity░index\n");
mprintf("░░░░Variable░X1░=░%f\n",polychaos_getindextotal(pc,1));
mprintf("░░░░Variable░X2░=░%f\n",polychaos_getindextotal(pc,2));
mprintf("░░░░Variable░X3░=░%f\n",polychaos_getindextotal(pc,3));
```

The previous script produces the following output.

```
Mean          = 3.500000
Variance      = 13.842473
First order sensitivity index
  Variable X1 = 0.313953
  Variable X2 = 0.442325
  Variable X3 = 0.000000
Total sensitivity index
  Variable X1 = 0.557675
  Variable X2 = 0.442326
  Variable X3 = 0.243721
```

We now focus on the variance generated by the variables #1 and #3. We set the group to the empty group with the `polychaos_setgroupempty` function and add variables with the `polychaos_setgroupaddvar` function.

```
groupe = [1 3];
polychaos_setgroupempty ( pc );
polychaos_setgroupaddvar ( pc , groupe(1) );
polychaos_setgroupaddvar ( pc , groupe(2) );
mprintf("Fraction░of░the░variance░of░a░group░of░variables\n");
mprintf("░░░░Groupe░X1░et░X2░=%f\n",polychaos_getgroupind(pc));
```

The previous script produces the following output.

```
Fraction of the variance of a group of variables
  Groupe X1 et X2 =0.557674
```

The function `polychaos_getanova` prints the functional decomposition of the normalized variance.

```
polychaos_getanova(pc);
```

The previous script produces the following output.

```
1 0 0 : 0.313953
0 1 0 : 0.442325
1 1 0 : 1.55229e-009
0 0 1 : 8.08643e-031
```

```

1 0 1 : 0.243721
0 1 1 : 7.26213e-031
1 1 1 : 1.6007e-007

```

We can compute the density function associated with the output variable of the function. In order to compute it, we use the `polychaos_buildsample` function and create a Latin Hypercube Sampling with 10000 experiments. The `polychaos_getsample` function allows to query the polynomial and get the outputs. We plot it with the `histplot` Scilab graphic function, which produces the figure 4.4.

```

polychaos_buildsample(pc,"Lhs",10000,0);
sample_output = polychaos_getsample(pc);
scf();
histplot(50,sample_output)
xtitle("Ishigami - Histogram");

```

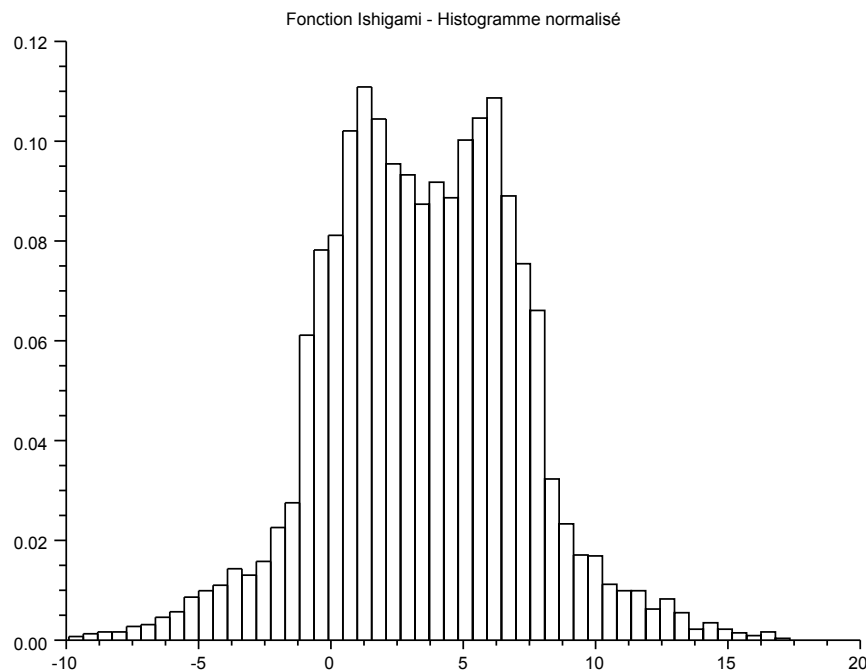


Figure 4.4: Ishigami function - Histogram of the output of the chaos polynomial on a LHS design with 10 000 experiments.

We can plot a bar graph of the sensitivity indices, as presented in figure 4.5.

```

for i=1:nx
    indexfirst(i)=polychaos_getindexfirst(pc,i);
    indextotal(i)=polychaos_getindextotal(pc,i);
end
scf();

```

```

bar(indextotal,0.2,'blue');
bar(indexfirst,0.15,'yellow');
legend(["Total" "First_order"],pos=1);
xtitle("Ishigami - Sensitivity indices");

```

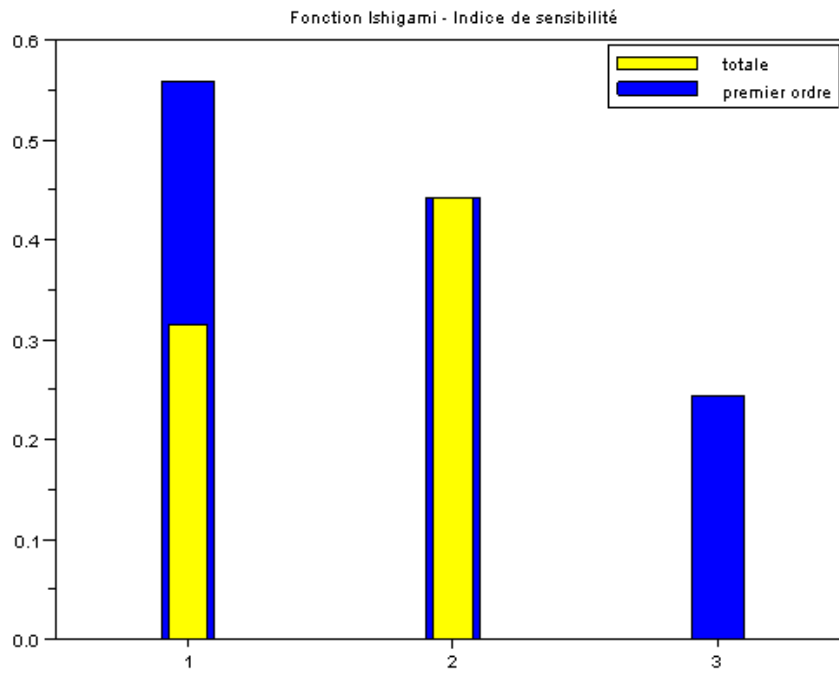


Figure 4.5: Ishigami function - Sensitivity indices.

Chapter 5

Thanks

Many thanks to Allan Cornet.

Bibliography

- [1] Michael Baudin. Programming in scilab, 2010.
- [2] Michael Baudin and Jean-Marc Martinez. Polynômes de chaos sous scilab via la librairie nisp. In *42èmes Journées de Statistique, 24 au 28 mai 2010 - Marseille, France*, 2010.
- [3] Didier Pelat. *Bases et méthodes pour le traitement des données (Bruits et Signaux)*. Master M2 Recherche : Astronomie-astrophysique, 2006.

Appendix A

Installation

A.1 Architecture of the toolbox

Let us present some details of the the internal components of the toolbox. The following list is an overview of the content of the directories:

- *tbxnisp/demos* : demonstration scripts
- *tbxnisp/doc* : the documentation
- *tbxnisp/doc/usermanual* : the L^AT_EXsources of this manual
- *tbxnisp/etc* : startup and shutdown scripts for the toolbox
- *tbxnisp/help* : inline help pages
- *tbxnisp/macros* : Scilab macros files *.sci
- *tbxnisp/sci_gateway* : the sources of the gateway
- *tbxnisp/src* : the sources of the NISP library
- *tbxnisp/tests* : tests
- *tbxnisp/tests/nonreg_tests* : tests after some bug has been identified
- *tbxnisp/tests/unit_tests* : unit tests

A.2 Installing the toolbox from the sources

In this section, we present the steps which are required in order to install the toolbox from the sources.

In order to install the toolbox from the sources, a compiler is required to be installed on the machine. This toolbox can be used with Scilab v5.1 and Scilab v5.2. We suppose that the archive has been unpacked in the "tbxnisp" directory. The following is a short list of the steps which are required to setup the toolbox.

1. build the toolbox : run the *tbxnisp/builder.sce* script to create the binaries of the library, create the binaries for the gateway, generate the documentation
2. load the toolbox : run the *tbxnisp/load.sce* script to load all commands and setup the documentation
3. setup the startup configuration file of your Scilab system so that the toolbox is known at startup (see below for details),
4. run the unit tests : run the *tbxnisp/runtests.sce* script to perform all unit tests and check that the toolbox is OK
5. run the demos : run the *tbxnisp/rundemos.sce* script to run all demonstration scripts and get a quick interactive overview of its features

The following script presents the messages which are generated when the builder of the toolbox is launched. The builder script performs the following steps:

- compile the NISP C++ library,
- compile the C++ gateway library (the glue between the library and Scilab),
- generate the Java help files from the .xml files,
- generate the loader script.

```
-->exec C:\tbxnisp\builder.sce;
Building sources...
  Generate a loader file
  Generate a Makefile
  Running the Makefile
  Compilation of utils.cpp
  Compilation of blas1_d.cpp
  Compilation of dcdflib.cpp
  Compilation of faure.cpp
  Compilation of halton.cpp
  Compilation of linpack_d.cpp
  Compilation of niederreiter.cpp
  Compilation of reversehalton.cpp
  Compilation of sobol.cpp
  Building shared library (be patient)
  Generate a cleaner file
  Generate a loader file
  Generate a Makefile
  Running the Makefile
  Compilation of nisp_gc.cpp
  Compilation of nisp_gva.cpp
  Compilation of nisp_ind.cpp
  Compilation of nisp_index.cpp
  Compilation of nisp_inv.cpp
  Compilation of nisp_math.cpp
```

```

Compilation of nisp_msg.cpp
Compilation of nisp_conf.cpp
Compilation of nisp_ort.cpp
Compilation of nisp_pc.cpp
Compilation of nisp_polyrule.cpp
Compilation of nisp_qua.cpp
Compilation of nisp_random.cpp
Compilation of nisp_smo.cpp
Compilation of nisp_util.cpp
Compilation of nisp_va.cpp
Compilation of nisp_smolyak.cpp
Building shared library (be patient)
Generate a cleaner file
Building gateway...
Generate a gateway file
Generate a loader file
Generate a Makefile: Makelib
Running the makefile
Compilation of nisp_gettoken.cpp
Compilation of nisp_gwsupport.cpp
Compilation of nisp_PolynomialChaos_map.cpp
Compilation of nisp_RandomVariable_map.cpp
Compilation of nisp_SetRandomVariable_map.cpp
Compilation of sci_nisp_startup.cpp
Compilation of sci_nisp_shutdown.cpp
Compilation of sci_nisp_verboselevelset.cpp
Compilation of sci_nisp_verboselevelget.cpp
Compilation of sci_nisp_initseed.cpp
Compilation of sci_randvar_new.cpp
Compilation of sci_randvar_destroy.cpp
Compilation of sci_randvar_size.cpp
Compilation of sci_randvar_tokens.cpp
Compilation of sci_randvar_getlog.cpp
Compilation of sci_randvar_getvalue.cpp
Compilation of sci_setrandvar_new.cpp
Compilation of sci_setrandvar_tokens.cpp
Compilation of sci_setrandvar_size.cpp
Compilation of sci_setrandvar_destroy.cpp
Compilation of sci_setrandvar_freememory.cpp
Compilation of sci_setrandvar_addrandvar.cpp
Compilation of sci_setrandvar_getlog.cpp
Compilation of sci_setrandvar_getdimension.cpp
Compilation of sci_setrandvar_getsize.cpp
Compilation of sci_setrandvar_getsample.cpp
Compilation of sci_setrandvar_setsample.cpp
Compilation of sci_setrandvar_save.cpp
Compilation of sci_setrandvar_buildsample.cpp
Compilation of sci_polychaos_new.cpp
Compilation of sci_polychaos_destroy.cpp

```

```

Compilation of sci_polychaos_tokens.cpp
Compilation of sci_polychaos_size.cpp
Compilation of sci_polychaos_setdegree.cpp
Compilation of sci_polychaos_getdegree.cpp
Compilation of sci_polychaos_freememory.cpp
Compilation of sci_polychaos_getdimoutput.cpp
Compilation of sci_polychaos_setdimoutput.cpp
Compilation of sci_polychaos_getsizetarget.cpp
Compilation of sci_polychaos_setsizetarget.cpp
Compilation of sci_polychaos_freememtarget.cpp
Compilation of sci_polychaos_settarget.cpp
Compilation of sci_polychaos_gettarget.cpp
Compilation of sci_polychaos_getdiminput.cpp
Compilation of sci_polychaos_getdimexp.cpp
Compilation of sci_polychaos_getlog.cpp
Compilation of sci_polychaos_computeexp.cpp
Compilation of sci_polychaos_getmean.cpp
Compilation of sci_polychaos_getvariance.cpp
Compilation of sci_polychaos_getcovariance.cpp
Compilation of sci_polychaos_getcorrelation.cpp
Compilation of sci_polychaos_getindexfirst.cpp
Compilation of sci_polychaos_getindextotal.cpp
Compilation of sci_polychaos_getmultind.cpp
Compilation of sci_polychaos_getgroupind.cpp
Compilation of sci_polychaos_setgroupempty.cpp
Compilation of sci_polychaos_getgroupinter.cpp
Compilation of sci_polychaos_getinvquantile.cpp
Compilation of sci_polychaos_buildsample.cpp
Compilation of sci_polychaos_getoutput.cpp
Compilation of sci_polychaos_getquantile.cpp
Compilation of sci_polychaos_getquantwilks.cpp
Compilation of sci_polychaos_getsample.cpp
Compilation of sci_polychaos_setgroupaddvar.cpp
Compilation of sci_polychaos_computeoutput.cpp
Compilation of sci_polychaos_setinput.cpp
Compilation of sci_polychaos_propagateinput.cpp
Compilation of sci_polychaos_getanova.cpp
Compilation of sci_polychaos_setanova.cpp
Compilation of sci_polychaos_getanovaord.cpp
Compilation of sci_polychaos_getanovaordco.cpp
Compilation of sci_polychaos_realisation.cpp
Compilation of sci_polychaos_save.cpp
Compilation of sci_polychaos_generatecode.cpp
Building shared library (be patient)
Generate a cleaner file
Generating loader_gateway.sce...
Building help...
Building the master document:
    C:\tbxnisp\help\en_US

```

```
Building the manual file [javaHelp] in
C:\tbxnisp\help\en_US.
(Please wait building ... this can take a while)
Generating loader.sce...
```

The following script presents the messages which are generated when the loader of the toolbox is launched. The loader script performs the following steps:

- load the gateway (and the NISP library),
- load the help,
- load the demo.

```
-->exec C:\tbxnisp\loader.sce;
Start NISP Toolbox
    Load gateways
    Load help
    Load demos
```

It is now necessary to setup your Scilab system so that the toolbox is loaded automatically at startup. The way to do this is to configure the Scilab startup configuration file. The directory where this file is located is stored in the Scilab variable `SCIHOME`. In the following Scilab session, we use Scilab v5.2.0-beta-1 in order to know the value of the `SCIHOME` global variable.

```
-->SCIHOME
SCIHOME =
C:\Users\baudin\AppData\Roaming\Scilab\scilab-5.2.0-beta-1
```

On my Linux system, the Scilab 5.1 startup file is located in

```
/home/myname/.Scilab/scilab-5.1/.scilab.
```

On my Windows system, the Scilab 5.1 startup file is located in

```
C:/Users/myname/AppData/Roaming/Scilab/scilab-5.1/.scilab.
```

This file is a regular Scilab script which is automatically loaded at Scilab's startup. If that file does not already exist, create it. Copy the following lines into the `.scilab` file and configure the path to the toolboxes, stored in the `SCILABTBX` variable.

```
exec("C:\tbxnisp\loader.sce");
```

The following script presents the messages which are generated when the unit tests script of the toolbox is launched.

```
-->exec C:\tbxnisp\runtests.sce;
Tests beginning the 2009/11/18 at 12:47:45
TMPDIR = C:\Users\baudin\AppData\Local\Temp\SCI_TMP_6372_
001/004 - [tbxnisp] nisp.....passed : ref created
002/004 - [tbxnisp] polychaos1.....passed : ref created
003/004 - [tbxnisp] randvar1.....passed : ref created
004/004 - [tbxnisp] setrandvar1.....passed : ref created
-----
```

Summary

tests	4	-	100	%
passed	0	-	0	%
failed	0	-	0	%
skipped	0	-	0	%
length			3.84	sec

Tests ending the 2009/11/18 at 12:47:48